UML MODELLING AND CODE GENERATION FOR AGENT-BASED, DISCRETE EVENTS SIMULATION

Giovanni A. Cignoni[a], Stefano Paci[b]

[a]Dipartimento di Informatica, Università di Pisa
[b]Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

[a]cignoni@di.unipi.it, [b]nefasto.cipa@yahoo.it

ABSTRACT
GeneSim is an open-source research project about software simulation of dynamic systems. The project focuses on UML as modelling language and on automated tools to generate the simulator from the UML model. This paper presents an implementation of this approach applied to *Agent-based Discrete Event* simulation. The system to be simulated is described by a set of UML diagrams which specifies an object-oriented model of the system. From such UML model it is possible to derive C++ source code by applying a set of defined code generation patterns. A C++ library provides the runtime environment that, compiled and linked with the generated code, results in the executable system simulator. The code generation process can be automatized: the UML diagrams are encoded in XML documents which are readable by an ad hoc compiler that uses XSLT transformations to actually generate the C++ code.

Keywords: UML, agent-based simulation, discrete event simulation, code generation.

1. INTRODUCTION
Advantages of system modelling and simulation are today well recognized (Thesen and Travis 1989). Thanks to the continuous improvement of software and hardware technologies, the practice of *software simulation* is constantly growing.

In a software model, each element of the system is represented by a data structure (a variable, an array, an abstract data type...), so the software simulator also mimics the structure the real system. The running simulation program dynamically evolves through a defined sequence of states as does the real system.

The system evolution may be modelled by considering only the discrete points in time – the *events* – when relevant changes occur (Sánchez 2007). While general, the *discrete-event* modelling approach is particularly well suited for simulation of manufacturing systems, transportation and communication networks, information processing systems and queuing systems (Schriber and Brunner 2007).

An *agent* is an identifiable component of the system capable to make decisions. An agent has a set of characteristics and rules which determine its behaviour. Agent-based modelling is an approach to simulation, generally used for systems suitable to be modelled using a continue time representation (Macal and North 2008).

GeneSim (Cignoni 2006) is an open-source research project about software simulation of dynamic systems. The project focuses on UML as modelling language and on automated tools to generate the simulator from the UML model of the system. The GeneSim modelling and simulation technique follows an *Agent-based Discrete Events* (ADE) approach.

We use UML to specify an object-oriented model of the system. In particular, the model is specified by UML class diagrams, object diagrams, state machine diagrams, and activity diagrams. Our technique defines the rules to build a complete model of the system, that is, a model that includes all the information needed to generate the simulator.

To actually generate the simulator, the UML model is encoded in XML respecting the syntax defined by a set of XML *schemas*. In such format the model is given as input to the compiler that generates the C++ source code of the simulator via XSLT transformations. The C++ code compiled and linked to a runtime library results in a ready-to-run simulator.

Our framework is suitable to be used for all those typical discrete-event simulation applications. Experiments were already carried out on real case studies involving public transportation networks (see section 6).

In Section 2 we resume the reference context of discrete event modelling and simulation. The GeneSim approach to ADE simulation is introduced in Section 3. In Section 4 we discuss a simple case study in order to introduce our UML notation. The code generation process is shown in Section 5. In Section 6 we present the case studies used to validate the framework.

2. DISCRETE EVENT SIMULATION
In order to understand the characteristics of the GeneSim approach we briefly summarize some classics of discrete event simulation.

Discrete Event systems Specification (DEVS), as proposed by Zeigler (1976), is a modelling formalism that has been used in many fields, ranging from parallel computing (Liu and Wainer 2010) to peer-to-peer network systems (Cheon, Seo, Park and Zeigler 2004) and, of course, simulation. A DEVS model is built by coupling several components known as *atomic models*. Each atomic model has a set of states and transition functions which describe its behaviour. Complex components are obtained by coupling atomic ones. This assembling technique helps the reuse of the model components (MacSween and Wainer, 2004).

As discussed by Pidd, Oses and Brooks (1999), a relevant DEVS issue is complexity. In fact, it is common to end with a model where each atomic model is coupled with almost every other, often resulting in unmanageable chains of dependencies.

Starting from the requirements defined by Nance (1977) and Sargent (1992) for a "next-generation modelling framework", Page (1994) claims that the DEVS-based approaches are able to describe systems only at low level. Page argues that DEVS has limited expressiveness, and lacks of traceability of the real system elements in the model components.

Hills and Poole (1969) described one of the first graphical notations for simulation: the *activity cycle diagrams* (ACD). Using this notation, the modeller is able to describe the system from different perspectives and to identify the *entities* that populate the system.

Modelling methods which use the ACD notation aim to describe the behaviour of the whole system as a sequence of operations that have to be atomized as analysis is refined. Each atomic operation, called *activity*, represents the consequences of a change of state in the system. Activities are linked together in a diagram which contains all the entities in the system.

One of the strengths of the ACD notation is its diffusion: it is long since it has been used as front-end for simulator generators and interpreters, like, for example, GASSNOL (Vidallon, 1980). Filho and Hirata (2004) also present an automatic translation from an extended ACD to a Java program. However, as Page already observed (1994), is not unusual that the code generated by these tools, in order to be actually executable, must be manually edited by the user. Moreover, modelling a system with ACD often results in a set of entities heavily-dependent on each other. As stated by Overstreet and Nance (1985) the resulting diagrams fail to represent the *conceptual model* of the system.

At run time the activities are stored in a *table*. The management cost of the activity table is yet another flaw of the simulation based on ACD. For this reason, Pidd (1992a, 1992b) suggested an optimization, called *three phase approach*, which also partially reduces dependencies among the entities of the system. However, the ACD model is still seen as a whole and its components are not truly independent. The behaviour of an entity can not be localized: it must be seen in the "great picture" of the system. Hence, many parts of the model are not reusable *as is* in other simulation projects.

As emerges from the above – pretty historical – summary of Discrete Event Modelling and Simulation, we can consider *re-usability*, *traceability* and *independence* of model components as three quality goals which had always driven the research, in particular they are the main objectives of our approach to ADE modelling and simulation.

3. GENESIM APPROACH

As model quality requirements, re-usability, traceability and independence are shared with other approaches to simulation. In particular, agent-based modelling sets the independence of the components as one of its starting assumptions (Macal and North, 2008).

The object-oriented (OO) paradigm can be exploited to enhance the traceability of the model components through different abstraction levels, even down to the source code of the simulator. OO paradigm is also suitable to introduce reuse mechanisms – like inheritance – in the modelling technique. SIMULA (Dahl and Nygaard, 1966) was a recognized forerunner in using the OO paradigm for the purposes of simulation.

In this section we introduce our approach to object oriented, agent-based, discrete event modelling and simulation. We then discuss how this approach is implemented in our framework.

3.1. An Object-oriented Agent-based Model

As defined by Macal and North (2008), an *agent* is an identifiable, individual component of the system that is capable to make decisions. An agent has a set of characteristics and rules which determine its own behaviour, therefore in the agent-based approach there is not a centralized management of the state of the system. In our ADE context, an agent can be considered as an *entity* that independently reacts to *events*.

According to the OO paradigm, we distinguish between entity *types* (classes in the OO terminology) and entity *instances* (objects). Each type defines a set of attributes (characteristics) and methods (rules). By inheritance it is possible to specialise previously defined entity types to reuse them in different models.

An entity type is *active* if it reacts to events. A *state machine* can be used to specify how each entity instance of a given active type reacts to events by changing its internal state. As part of the definition of an entity type, the state machine can be inherited from a superclass or overridden by subclasses. An *agent* is an instance of an active entity type.

Passive entity types are also defined as, in practice, they may be useful to represent data structures. Passive entity instances does not react to events, however they are still able to interact with other model components.

3.2. Discrete Events

According to classic discrete-event modelling, events are the points in time in which the system state changes. In our approach we see events as facts that happen at given instants and cause changes in the state of agents. This perspective makes possible to:

- separately identify facts that happen at the same instant and select them in priority order;
- let the events be managed by the only agents actually affected by them;

- consider the system state as composition of the states of its entities.

As for entities, there are event *types* and event *instances*. Event types are referred by the state machines which specify the behaviour of entity types, while during the simulation actual agents react to event instances.

Each event instance needs to know which agents have to handle it, and in what order. There is no event dispatcher or centralized event-dispatching algorithm: our approach introduces *self-dispatching* events. That is, like entity instances, also event instances are independent components of the system with their own dispatching behaviour. From this perspective, our approach is further characterised as agent-based.

Event instances are usually scheduled dynamically as part of the behaviour of the agents. Events instances may also be part of the model, for instance to specify the initial events needed to start the simulation.

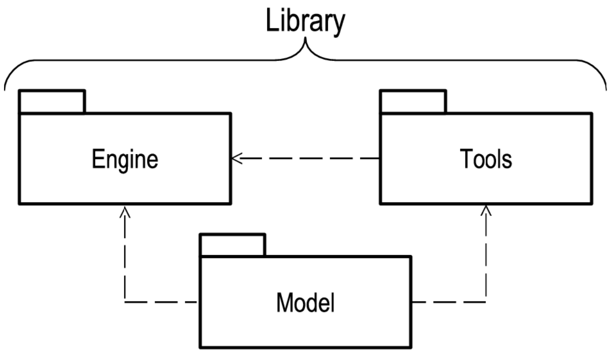

Figure 1: Generic Software Architecture of a Simulator

3.3. Simulation Environment

Fig.1 is an UML *package diagram* representing the general architecture of a software simulator. As the diagram shows, modelling a system in a given software simulation framework depends on the library components: the modeller must usually be aware of features and characteristics of the runtime platform.

In our environment the simulation e*ngine* and the utility *tools* are provided by the *GS_DSLibs* C++ runtime library. The simulation engine is an event agenda, which does – conceptually – a very simple work:

1. advances the current time to next event;
2. triggers all the events that are scheduled at the current time;
3. repeat from 1 until no scheduled events are present in the agenda.

When events are scheduled at the same time, they are triggered in priority order, which is an optional attribute for every derived event class. Once triggered, an event instance follows its own dispatching behaviour to interact with its handling agents.

The GS_DSLibs engine package provides other features like the ability to *withdraw* events previously scheduled in the agenda. This can be useful in many practical situations involving behaviour branches, for instance the management of time-out events.

The GS_DSLibs tools package includes utility classes for simulation: data structures like queues and sets and pseudo-random number generators for several statistical distributions.

The third package of the general architecture of a software simulator is the implementation of the model of the system. In our environment this is the actual part needed in order to generate the simulator.

In many simulation environments the implementation of the model is just a collection of data suitable to be interpreted on the fly by an ad hoc engine. In our approach the implementation of the model is obtained directly from its UML specification which includes the entity and event types, and the initial set of their instances that determines the state of the system when the simulation starts.

In order to model a system there is no need of a deep knowledge of the GS_DSLibs library packages: the links with the runtime components are discreetly managed by the code generator when it compiles the UML model.

4. UML

In this section we introduce the UML notations that are used for ADE modelling in our approach. The presentation follows a simple case study.

4.1. Using UML in the Simulation Process

The case study is presented by introducing the diagrams that make up the final model. This “waterfall” presentation does not match the practice of modelling. Generally, the model diagrams are refined through successive iterations of analysis steps.

During these iterations, the modeller may use UML in different ways. Fowler (2003) described three levels of using *UML: sketch*, *blueprint* and *program*.

Sketch is typical in the early phases of the analysis, when the general choices of the model are discussed. Diagrams at the blueprint-level are aimed to *forward engineering*, that is, to build a detailed design of the model before implementing it. These UML practices, while originated for software engineering, well apply also to system modelling for simulation.

As far as design decisions, a blueprint model is complete, but is not sufficient to fully specify the model for the simulator generation: it has to be integrated with some source code. For instance, we might need to specify methods that detail the “nuts and bits” of the behaviour of agents and events. Here is where other techniques rely on manual code completion. Our method permits to complete the model diagrams still using the UML syntax at the program-level.

In this section we present the diagrams at the blueprint-level, an example of a complete program-level diagram will be shown in section 5.

4.2. Description of the System

As a case study we use the *Harassed Clerk* example first proposed by Pidd (1992b). The system involves a clerk and two kinds of clients: those arriving at the service desk, and those calling on the phone. The system is described as follows:

1. there is only one clerk in the system, serving one client at a time;
2. arriving (or calling) clients that find the clerk busy wait in two first-in-first-out queues, one for each client type;
3. each client (of both types) is characterized by its arrival time and its service time;
4. queues have no length limits;
5. the clerk works without interruptions;
6. desk clients have higher priority;
7. the clients wait until they are served, no matter how much time it requires.

4.3. Identifying the Entities

There are three entity types: the *clerk,* the *desk client* and the *phone client*. There is just one clerk agent and an undetermined number of client agents, of both types.

We consider all entity types as active entities: their instances will react to events. Therefore, we build a hierarchy of classes all deriving from the root library class *ActiveEntity*.

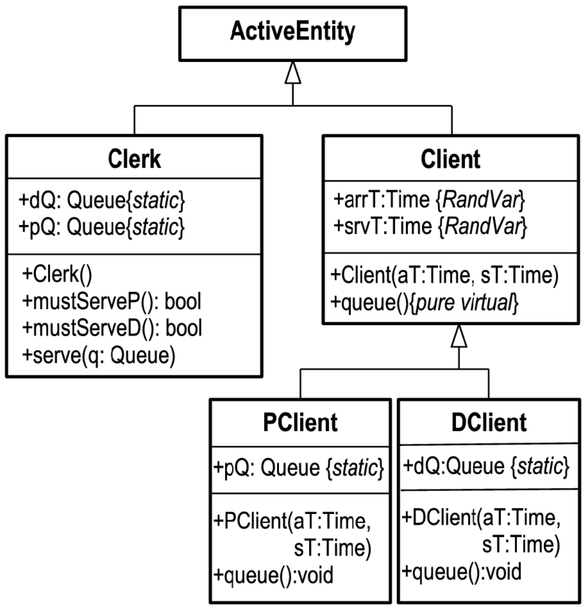

Figure 2: Classes of System Agents, Blueprint-level

The class specification introduces the attributes whose values will characterize the agents during the simulation. Few methods are introduced to improve readability of the transition arcs of the state machine diagrams (see Section 4.6). In fig. 2 we can observe:

- *Client* has two *Time* attributes to specify arrival time and service time;
- the *Client* class cannot be instantiated, because *queue* is *pure virtual*, and its model does not include a state machine (see Section 4.6);
- both *PClient* and *DClient* have a constructor method which takes two Time arguments (arrival time and service time) and calls the constructor of the *Client* parent class; the *Clerk* constructor has no parameters;
- both *PClient* and *DClient* have a static *Queue** attribute to refer their own queues; *Clerk* has static references to both the queues;
- *Client* has a method for queuing, while *Clerk* has one to implement the serving procedure;
- *Clerk* has two methods that implement the priority policy, *mustServeP* and *mustServeD.*

The *arrT* and *srvT* attributes are random variables: according to proper distributions, they get different values for each *Client* agent. The implementation of random variables is provided by the GS_DSLibs runtime library, however we will not discuss the details here.

Every active entity class must have (or inherit) an *handle(Event*)* method that specify how its agents react to event instances. In our approach the handle method is specified by a state machine diagram associated to each active entity class (see Section 4.6).

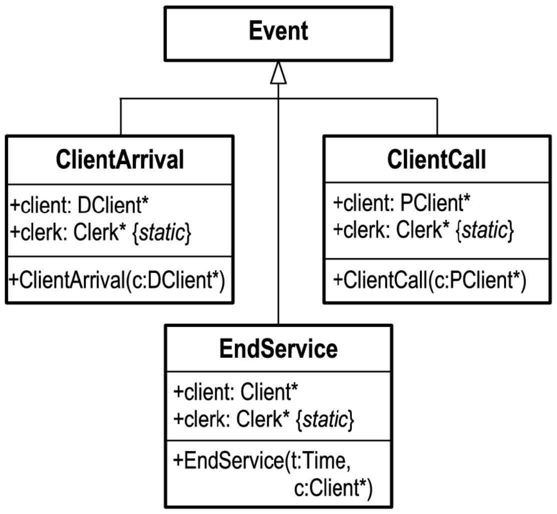

Figure 3: Classes of System Events, Blueprint-level

4.4. Identifying the Events

There are three event types:

- *arrival of a client*, which starts the life of a new *DClient* and, depending on the state of the clerk (busy or not), may start a service or a wait in the desk client queue;
- *client call request*, which starts the life of a new *PClient* and, depending on the state of the clerk (busy or not), may start a service or a wait in the phone client queue;
- *end of a service*, which ends the life of a desk or phone client and, depending on the state of

the queues (empty or not), may let the clerk go idle or serve another client.

As shown in Fig. 3, each event has two attributes: a static pointer to the only *Clerk* instance in the system, and a pointer to the *Client* agent the event relates to.

The *Event* subclasses must implement the *doAction()* method, which is modelled by an activity diagram (see Section 4.5). The purpose of *doAction()* is to specify how the event will be dispatched to the agents that have to handle it. The *doAction()* method call, done by the simulator engine, starts the event triggering.

The *Event* root library class defines two attributes, *time* and *priority,* needed by the engine to order the events in the agenda. In our case study, the values of time and priority are set by the constructors of the subclasses. For instance, *ClientArrival* sets its time value equal to the *arrT* attribute (see Fig. 2) of the *DClient* argument of the constructor.

4.5. Linking Events to Agents

As shown in Fig. 4, we use UML activity diagrams to specify the dispatching behaviour of events. In practice, the activity diagram describes the algorithm used to call the *handle(Event*)* methods of the agents affected by the event. This mechanism assures that only these agents are involved in the handling of a specific event and in the right order.

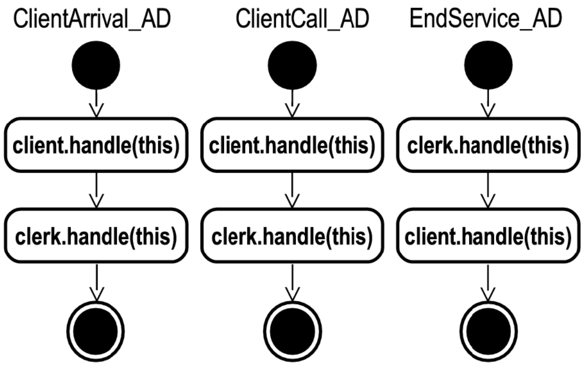

Figure 4: Event Self-dispatching

In this case study we have very simple *doAction()* methods. Using activity diagrams to specify a mere sequence seems overkilling. However, activity diagrams are useful in those cases where there is need to express more complex algorithms. For instance when the agent that has to manage an event must be chosen among a set of available ones by applying a given policy.

Note that for *ClientCall* and *ClientArrival*, the sequence specified by the activity diagrams is relevant: these events must be handled first by the *Client* agent, in this way it is ready to be popped from the queue by the *Clerk* agent (if idle, see Fig. 5, Fig. 6 and Fig. 7).

4.6. Specifying How Agents React to Events

The model of an entity type includes a state machine diagram to specify *handle(Event*)*, that is, how an agent of that type, depending on its current state and custom conditions, reacts to events performing actions and, maybe, going in a different state.

In our case study, the *DClient* and *PClient* state machines have three states (see Fig. 5 and Fig. 6): *arriving* (or *dialing* for *PClient*), *inService*, and *served*. Note that, in order to be minimal, *inService* merges two different conditions: the client waiting its turn and the client being actually served.

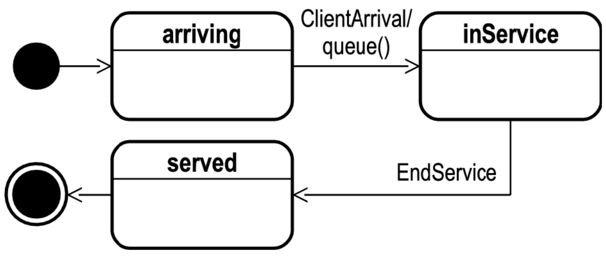

Figure 5: *DClient* Behaviour

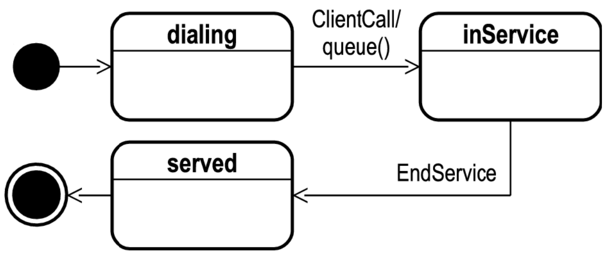

Figure 6: *PClient* Behaviour

As shown in Fig. 5 *arriving* is marked as the initial state: a *DClient* agent is created in this state; *served* is marked as the final state: as a *DClient* enters this state it is destroyed. Our UML notation consider start and final icons as labels instead of states. As a consequence the linked transitions are *completion transition* as defined by OMG (2011). This notation choice ables the sound interpretation of diagrams by the code generator.

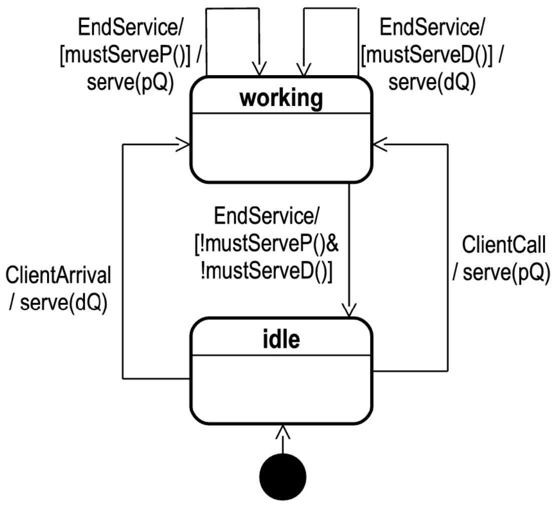

Figure 7: *Clerk* Behaviour

A *DClient* stays in the initial state until a *ClientArrival* event happens: at that time the *DClient* agent performs the associated action and goes in the *inService* state. It reaches its *served* final state when, being in the *inService* state, its *EndService* event happens. The *PClient* state machine diagram in Fig. 6 is similar.

A *Clerk* agent (see Fig. 7) starts *idle*. When a *ClientArrival (or ClientCall)* happens it calls the *serve* method and goes *working*. While *working*, when an *EndService* happens the clerk evaluates its policy: if *mustServeP()* is *true* the clerk serves the phone queue, if *mustServeD()* is true it serves the desk queue; if both are false, it turns back *idle.* By definition, the two conditions will never be both true (see Section 5).

4.7. Initial State of the System

Before the simulation starts the system must be initialized: some agents have to be created, and some events must be scheduled in the agenda.

The initial state is specified by two UML object diagrams: one contains those object that must be created in every simulation, while the other is needed to specify how a particular simulation is initialized.

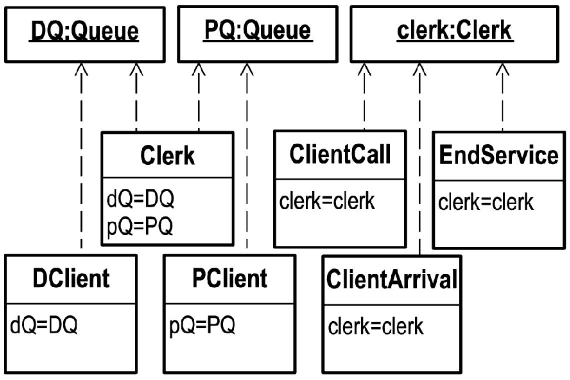

Figure 8: Initial State, Static Objects

The diagram in Fig. 8 shows the initializations that must be performed before every simulation starts. There is a *Clerk* agent and two *Queue* objects, which are stored as static attributes of the five classes of the model. The dependencies in the diagram mean that the *Clerk* class static references to the *Queue* objects must be set after the creation of the queues. The *Event* classes have to set their *Clerk* static pointer after the construction of the *Clerk* agent, as expected.

Our system consists of four Client agents and the initial agenda contains two instances of *ClientArrival* and two instances of *ClientCall* (see Fig. 9). This last diagram may differ in every simulation experiment, as the user may want to observe different client sequences.

When large numbers of transient objects are involved, it is convenient to model and implement agents which play as event/agent generators. On the other hand, a diagram as the one depicted in Fig. 9 is typical for validation purposes: simple configurations are needed to test the model using ad hoc initial states which produce a predictable output.

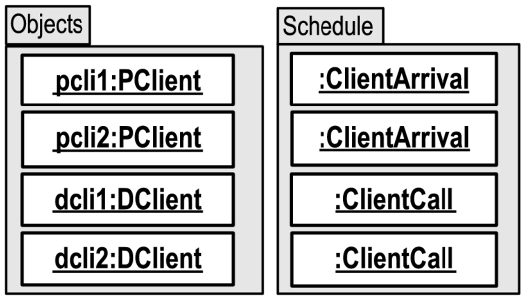

Figure 9: Initial State, Transient Objects

4.8. The Project Diagram

The project diagram is an index of the whole model, needed to give structure to all the specification.

It is an UML object diagram (see Fig. 10) containing the packages for *entities* and *events*. Inside those packages we distinguish between elements that specify either the *structure* or the *behaviour* of an entity or an event. A third packages is the *system initial state*, which includes the diagrams described in section 4.7.

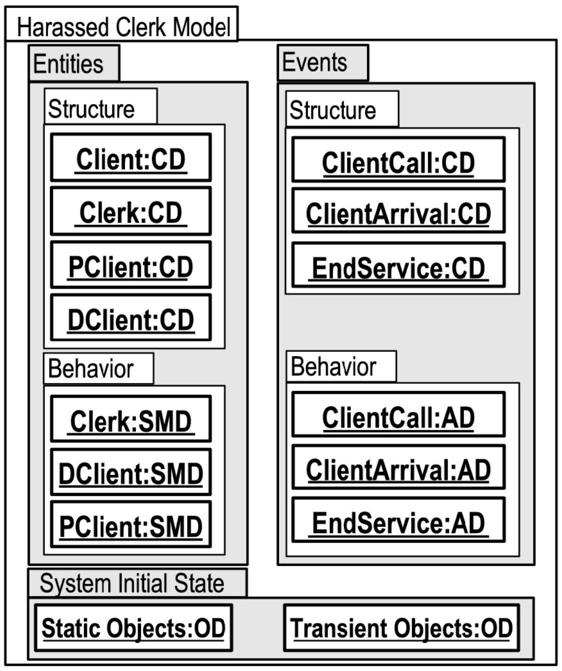

Figure 10: Project Diagram

4.9. Well-formed Models

In this section we propose a definition of “well-formed model”. This definition is exploited by the code generator to perform consistency checks on the model.

A model can be defined in set-theoretic terms as:

$M=<\mathfrak{A},\mathfrak{E},\mathfrak{B},\mathfrak{D},\mathfrak{O},\mathfrak{P}>$,

where:

- $\mathfrak{A}$ is a set of agent types (class diagrams),
- $\mathfrak{E}$ is a set of event types (class diagrams),

- $\mathfrak{B}$ is a set of agent behaviour specifications (state machine diagrams),
- $\mathfrak{D}$ is a set of event dispatching specifications (activity diagrams),
- $\mathfrak{O}$ is a set of object diagrams,
- $\mathfrak{P}$ is a set of project diagrams.

Definition: *Let $a,b \in A$, we say that a* ***inherits*** *from b, and write $a \to b$, if a is a diagram of a subtype of b. We say that a is a* ***child*** *of b, and b is a* ***parent*** *of a.*

The definition also applies to the elements of $\mathfrak{E}$.

Definition: *Let A be a subset of $\mathfrak{A}$, and $a_M \in A$, we define a_M as a* ***maximum*** *element in A if*
$\neg\exists\, a \in A : a_M \to a$,

Definition: *$a_m \in A$ is a* ***minimum*** *in A if*
$\neg\exists\, a \in A : a \to a_m$.

Definition: *Let A be a subset of $\mathfrak{A}$, not empty. A is defined as a hierarchy-connected set* ***(hc set)*** *if A has a maximum a_M and*
$\forall\, a \in A - \{a_M\} : \exists\, b \in A : a \to b$.

Definition*: $A \subset \mathfrak{A}$, A an hc set. A is a* ***linear set*** *if*
$\exists!\, a_m \in A$ *minimum for A.*

The definitions of hc set and linear set can be easily extended to subsets of $\mathfrak{E}$.

Definition*: Let $b \in \mathfrak{B}$ and $a \in \mathfrak{A}$. We say that b is* ***associated*** *to a, and write $b \mu a$, if b is the behaviour of a. Let $d \in \mathfrak{D}$, and $e \in \mathfrak{E}$, We say that d is associated to e, and write $d \mu e$, if d specifies the dispatching of e.*

Definition*: Let $d \in \mathfrak{D}$ and $e \in \mathfrak{E}$, $d \mu e$. We define the* ***handler set*** *of d, and write h(d), as a subset of A such that $\forall a \in h(d)$, a is the diagram of the entity type of an handler of the event e.*

Definition*: $b \in \mathfrak{B}$, $a \in \mathfrak{A}$, $b \mu a$, we define the* ***handled set*** *of b, written H(b), as a subset of $\mathfrak{E}$ such that*
$\forall\, e \in H(b)$, $a \in h(d)$ where $d \mu e$.

The definition of handler set and handled set should be extended to cover those cases where an entity a, included in h(d), does not define a state machine. In this case is correct to include the event e (such that $d \mu e$) in the handled set of the behaviours of the children of a.

Definition: Let M be a model. M is ***well-formed*** *if*

- *All $\mathfrak{A},\mathfrak{E},\mathfrak{B},\mathfrak{D},\mathfrak{O},\mathfrak{P}$ are not empty*
- $\#\mathfrak{O} = 2$
- $\#\mathfrak{P} = 1$
- $\forall\, b \in \mathfrak{B} : \exists!\, a \in \mathfrak{A} : b \mu a$
- $\forall\, d \in \mathfrak{D} : \exists!\, e \in \mathfrak{E} : d \mu e$
- $\forall A \subset \mathfrak{A}$ *linear hc set,* $\exists\, b \in \mathfrak{B} : \exists\, a \in A,\ b \mu a$
- $\forall E \subset \mathfrak{E}$ *linear hc set,* $\exists\, d \in \mathfrak{D} : \exists\, e \in E,\ d \mu e$
- $\forall b \in \mathfrak{B},\ \forall e \in H(b)$*, if* $b \mu a$ *and* $d \mu e$*, then* $a \in h(d)$
- *all the entity and event types used in the diagrams of $\mathfrak{O}$ must be defined in $\mathfrak{A},\mathfrak{E},\mathfrak{B},\mathfrak{D}$.*

5. CODE GENERATION

The ADE simulation framework includes a prototypical code generator, *GS_ADECodeGen.* The generator expects the diagrams to be encoded in a given XML format. The resultant C++ project is ready to be compiled, without further editing of the code, and retains the modular structure of the UML model. The intervention in the generated code is avoided because is possible to include the few really needed lines of C++ code in the UML model, as we discuss in this section.

After an overview of the code generation process (Section 5.1) we present a completed program-level diagram (Section 5.2), the XML encoding of the *PClient* class and state machine diagrams (Section 5.3) and the corresponding C++ code (Section 5.4).

5.1. The Code Generation Process

The development of a simulator using the ADE method is an example of *model-driven programming.* As stated by Sarkar (2002), XML/XSLT-based source code generation is a convenient paradigm, much simpler to use than Compiler-Compilers tools.

The UML diagrams and their XML encoding are two equivalent ways to specify the model, one is the human readable graphical form, the other is needed to feed the code generator.

The XML Schemas and Stylesheets are fixed inputs in the generation process. To parse and manipulate the XML documents we use the DOM interface provided by the Xerces library. Before the transformation, we need to check the model for semantic errors, and perform some manipulation (like merging structure and behaviour diagrams). The transformation process uses the Xalan library. Xerces and Xalan are open source projects developed by the Apache Foundation (1999).

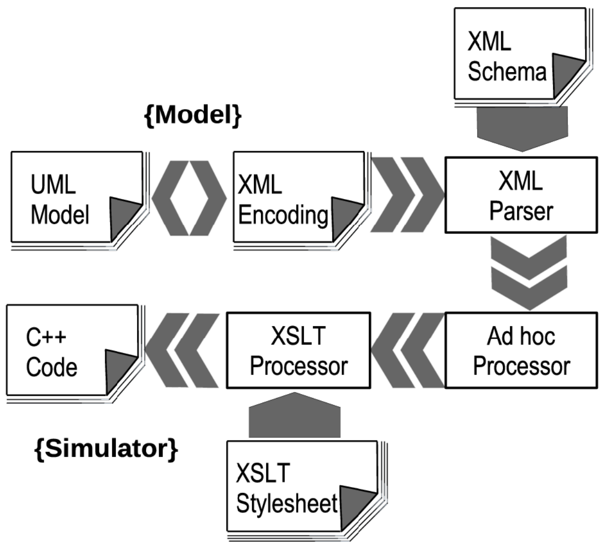

Figure 11: Code Generation Process

5.2. Completing the Diagrams

In order to generate the code, the blueprint-level UML model has to be completed to program-level. Our UML notation provide the conventions needed to integrate the missing code in the diagrams.

Fig. 12 shows a program-level version of the entity diagram in Fig. 2. The needed C++ code is included in comment boxes associated to the methods definition: *Clerk::serve(Queue*)* pops a client from the queue and schedule an *EndService* event. To schedule an event, each class derived from *ActiveEntity* inherits the *schedule(Event*)* method, which adds an event to the agenda. The *PClient* and *DClient* constructors use the *parent* stereotype, which indicates the superclass method call.

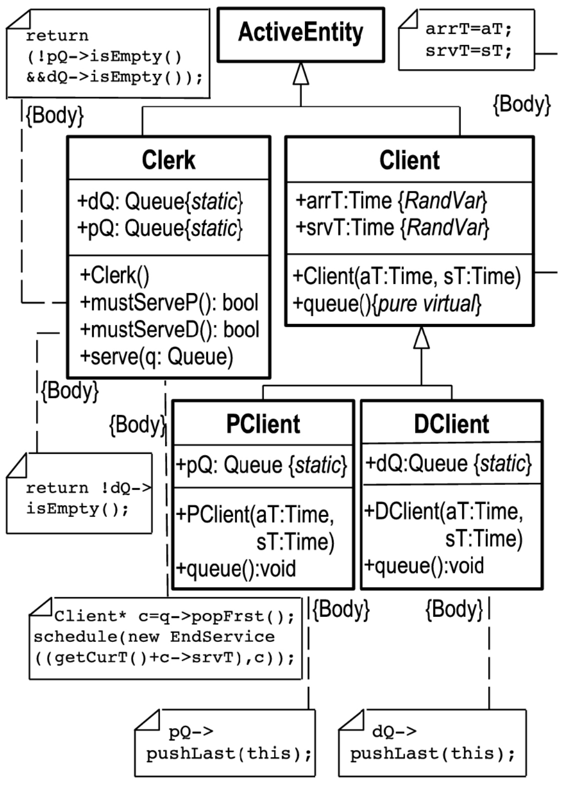

Figure 12: Classes of System Agents, Program-level

5.3. XML Encoding

In the following we show two excerpts of the XML files which encode the UML model of the case study. In particular, we present the XML encoding of the class and state machine diagrams for the PClient entity type.

In order to highlight the equivalence between the UML format and its XML encoding, the bold typeface is used to refer the XML code directly derived from the UML diagrams. Plain typeface is used for the needed XML syntax "glue".

5.3.1. PClient Entity Diagram

In the following we present the XML encoding of the PClient class diagram (see Fig. 12).

```
<entity name="PClient">
<relationship type="generalization">
    <class name="Client"/>
</relationship>
<encapsulation type="public">
<attribute name="pQ" type="GS_Queue*"
                static="true" default="NULL"/>
<method name="queue" type="void">
    <body text="pQ->pushLast(this);"/>
</method>
<method name="PClient" type="">
    <parentmethod name="Client">
    <parameter name="aT"/>
    <parameter name="sT"/>
    </parentmethod>
    <parameter name="aT" type="GS_Time"/>
    <parameter name="sT" type="GS_Time"/>
    <body text=""/>
</method>
</encapsulation>
</entity>
```

Methods and attributes are collected by encapsulation, as we are used to see in C++ header files. The inheritance relationship of *PClient* from *Client* is explicitly encoded as well it is the inheritance of *Client* from *ActiveEntity*, which implies that all its subclasses are active entities.

5.3.2. PClient State Machine Diagram

In the following we present the XML encoding of the *PClient* state machine diagram (see Fig. 6).

```
<StateChart activeEntityId="PClient">
    <state id="dialing" type="initial">
        <transition nextState="inService"
                    event="ClientCall"
                    action="queue()"/>
    </state>
    <state id="inService" type="regular">
        <transition nextState="served"
                    event="EndService"/>
    </state>
    <state id="served" type="final"/>
</StateChart>
```

5.4. C++ Code

In this section we show an examples of generated C++ code. In particular, we present the *PClient* sources. They are organized in one header file (.hpp) and one implementation file (.ccp).

In order to reveal some details of the generation process, the code that is obtained by verbatim replication of information already explicit in the XML encoding of the UML model is rendered in bold typeface. Plain typeface is used for the code added by application of the code generation patterns.

5.4.1. PClient.hpp

In the following we present the generated header file of the *PClient* class.

```
#include "../../../../GS_DSLibs-3.5b/
        GS_EvEng/lib/GS_EvEng.hpp"
#include "Client.hpp"
class ClientArrival;
class ClientCall;
class Clerk;
class DClient;
class EndService;
#define PCLIENT_DIALING 1
#define PCLIENT_INSERVICE 2
#define PCLIENT_SERVED 3
```

```
class PClient : public Client {
    private:
        int state;
    protected:
        static GS_Queue* pQ;
        void queue();
    public:
        PClient(GS_Time aT,GS_Time sT);
        virtual void handle(GS_Event* event);
        static void setpQ(GS_Queue* arg);
};
```

The .hpp file contains all the definitions given in the class diagram. The *handle(Event*)* method does not appear in the class diagram: it is added because in the model there is a state machine for *PClient* (see Fig. 10). Still from diagrams are derived: the setter for the static queue and the forward declarations of all the classes in the model (Fig. 2 and 3), a private attribute *state* and the defines for the identifiers of the entity states (Fig 6).

5.4.2. PClient.cpp

In the following we present the generated implementation file of the *PClient* class.

```
#include "PClient.hpp"
#include "../../../../GS_DSLibs-3.5b/
        GS_EvEng/lib/GS_EvEng.hpp"
#include "ClientCall.hpp"
#include "ClientArrival.hpp"
#include "EndService.hpp"
#include "Clerk.hpp"
#include "DClient.hpp"
GS_Queue* PClient::pQ=NULL;

void PClient::queue(){
    pQ->pushLast(this);
}

PClient::PClient(GS_Time aT,
                 GS_Time sT): Client(aT,sT) {
    state=PCLIENT_DIALING;
}

void PClient::setpQ(GS_Queue* arg) {
    pQ=arg;
}
```

Note that in the constructor the generator added the initialization of the *state* attribute, with the value of the PCLIENT_DIALING initial state.

```
void PClient::handle(GS_Event* event) {
    switch (state) {
    case PCLIENT_DIALING:
        if(event->getId()==CLIENTCALL) {
            queue();
            state=PCLIENT_INSERVICE;
        }
    break;
    case PCLIENT_INSERVICE:
        if(event->getId()==ENDSERVICE) {
            state=PCLIENT_SERVED;
        }
    break;
    case PCLIENT_SERVED:
        delete this;
    break;
    default:
        cerr << "Error: undefined state.\n";
    }
}
```

The code of the *handle* method is derived following a pattern from the state machine diagram:

- the switch-case construct is used to decide what is the current state of the entity;
- if-then-else selects the transition evaluating the specified guards;
- inside the body of the conditional branches is executed the specified action and the actual state transition.

6. EXPERIMENTAL USE AND FUTURE WORK

In this paper we presented the GeneSim approach to ADE simulation of dynamic systems. To ease the modelling process we exploit UML at different levels of detail. Moreover, we provide an efficient and safe transition from the UML model to the simulator (Paci 2011). The UML at its highest level of detail fully specifies the system and can be used as input to a C++ code generator. The resulting source code can be compiled and linked to our runtime library to obtain the simulator executable. Several components of this framework has been validated in real case studies.

The UML notation and the code generation patterns have been tested in a case study about the simulation of a demand responsive public transport system. The case study involved the province of Brescia and MAIOR srl, a leader firm in Italy for transport management software (Bertuccelli 2007, Gerardi 2007).

The runtime library has also been successfully tested in a case study about the simulation of the public bus service in La Spezia. The experiment was mainly targeted to validate the performance on a real size example: the transportation network counted 3071 vertices and 4300 links, the service schedule was made of 112 routes and 3029 daily courses. Simulation of a full day service required less than 1 sec on ordinary hardware. Moreover, the experiment showed that times are linear with the size of the service (Gervasi 2010, Cignoni and Gervasi 2011). The case study involved ATC Servizi Spa, the public company running the bus service in La Spezia, and, again, MAIOR Srl.

Future development of the project will improve our UML notation to explicit the scheduling of events. Furthermore, we plan to introduce an UML notation to represent object construction, maybe inspired to *normal object form* as proposed by Bunse and Atkinson (1999).

Another important development of the GeneSim environment for ADE modelling and simulation will be an UML editor capable of producing the XML encoding of the model. This tool will be syntax-driven, and should be developed together with further releases of the generator (GS_ADECodeGen). Future release of the code generator will also perform consistency checks on the model, and produce error and warning messages that would guide the user in the model building process.

REFERENCES

Bertuccelli, F., 2007. *Interfacce di acquisizione e analisi dati per la simulazione di sistemia di trasporto pubblico a chiamata*. Thesis (Bhc). Università di Pisa.

Bunse, C. and Atkinson, C., 1999. The normal object form: bridging the gap from models to code. *Proceedings of the 2nd international conference on The unified modelling language: beyond the standard,* pp. 675-690. Fort Collins (CO, USA).

Cheon, S., Seo, C., Park, S., and Zeigler, B. P., 2004. Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System . *Proceedings of the 2004 Advanced Simulation Technologies Conference - Design, Analysis, and Simulation of Distributed Systems (ASTC'04)*. April, Arlington (Virginia, USA).

Cignoni, G.A., 2006. *GeneSim Project Website*. Available at http://genesim.sourceforge.net [accessed 15 May 2012].

Cignoni, G.A., Gervasi, C., 2011. GS_DTLib: simulazione efficiente di sistemi di trasporto. *MobilityLab* 39.

Dahl, O. and Nygaard, K., 1966. SIMULA: an Algol-based Simulation Language. *Communication of the ACM* 9:671-678.

De Lara Araujo Filho, W. and Hirata, C.M., 2004. Translating Activity Cycle Diagrams to Java Simulation Programs. *ANSS '04 Proceedings of the 37th annual symposium on Simulation,* pp 157-164. April 18-22, Washington, DC (USA).

Fowler, M., 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd ed. Addison-Wesley.

Gerardi, L., 2007. MAIOR e la ricerca scientifica al servizio del trasporto flessibile. *MobilityLab* 15.

Gervasi, C., 2010. *Una libreria C++ per la simulazione a eventi discreti di sistemi di trasporto*. Thesis (BSc). Università di Pisa.

Hills, B.R. and Poole, T.G., 1969. A Method for Simplifying the Production of Computer Simulation Models . *TIMS Tenth American Meeting*. October 1-3, Atlanta (Georgia, USA).

Liu, Q. and Wainer, G.A., 2010. Accelerating Large-scale DEVS-based Simulation on the Cell Processor. *Proceedings of the 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium.* April, San Diego (California, USA).

Macal C.M. and North M.J., 2008. Agent-based modelling and simulation: ABMS examples. *Proceedings of the 2008 Winter Simulation Conference,* pp 101-112. December 7-10, Miami (Florida, USA).

MacSween P. and Wainer, G. A., 2004. On the Construction of Complex Models Using Reusable Components. *Proceedings of SISO Spring Simulation Interoperability Workshop.* Arlington (Virginia, USA).

Nance, R. E., 1977. *The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models.* Final Report to the National Bureau of Standards. Department of Computer Science, Virginia Tech, Blacksburg, VA, June.

Object Management Group, 2011. *OMG Unified Modeling Language (OMG UML), Superstructure*. 2.4.1.

Overstreet, C. M. and Nance, R. E., 1985. A specification language to assist in analysis of discrete event simulation model. *Communications of the ACM* 28: 190-201.

Paci, S., 2011. *Da UML a C++. Modellazione e generazione di codice per la simulazione ad eventi discreti*. Thesis (BSc). Università degli Studi di Firenze.

Page, E. H. Jr., 1994. *Simulation modelling methodology: principles and etiology of decision support*. Thesis (Ph.D.). Virginia Polytechnic Institute and State University.

Pidd, M., 1992a. *Computer Simulation in Management Science.* John Wiley & Sons.

Pidd, M., 1992b. Object Orientation & Three Phase Simulation. *Proceedings of the 1992 Winter Simulation Conference,* pp. 689-693. December 13-16, Arlington (Virginia, USA).

Pidd, M., Oses, N. and Brooks, R. J., 1999. Component-based simulation on the Web? *Proceedings of the 1999 Winter Simulation Conference,* pp. 1438-1444. December 5-8, Phoenix (Arizona, USA).

Sánchez, P.J., 2007. Fundamentals of simulation modelling *Proceedings of the 2007 Winter Simulation Conference,* pp. 54-62. December 9-12, Washington, DC (USA).

Sargent, R.G., 1992. Requirements of a Modeling Paradigm. *Proceedings of the 1992 Winter Simulation Conference*, pp. 780-782. December 13-16, Arlington (Virginia, USA).

Sarkar, S., 2002. *Model-driven programming using XSLT.* XML Journal, SYS-CON Media, Inc.

Schriber, T.J. and Brunner, D.T., 2007. Inside discrete-event simulation software: how IT works and why IT matters. *Proceedings of the 2007 Winter Simulation Conference,* pp. 113-123. December 9-12, Washington, DC (USA).

The Apache Software Foundation. (1999) *Apache Software Foundation – Projects Website*. Available at http://projects.apche.org [accessed 15 May 2012].

Thesen, A., Travis, L. E., 1989. Simulation for decision-making: an introduction. *Proceedings of the 1989 Winter Simulation Conference,* pp 9-18. December 4-6, Washington, DC (USA).

Vidallon C., 1980. GASSNOL: A computer subsystem for the generation of network oriented languages with syntax and semantic analysis. *Simulation '80*. June 25-27, Interlaken (Switzerland).

Zeigler B.P., 1976. *Theory of modelling and simulation*. John Wiley & Sons.